AFRL-RV-PS-
TR-2014-0035

THE CHALLENGES OF SENSING AND REPAIRING SOFTWARE DEFECTS IN AUTONOMOUS SYSTEMS

Stephanie Forrest and Westley Weimer

Regents of the University of New Mexico
MSC01 1247
1 University of New Mexico
Albuquerque, NM 87131-0001

9 May 2014

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

DTIC COPY
NOTICE AND SIGNATURE PAGE

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RV-PS-TR-2014-0035 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//SIGNED//
LESLIE VAUGHN
Program Manager

//SIGNED//
PAUL D. LEVAN, Ph.D.
Technical Advisor, Space Based Advanced Sensing and Protection

//SIGNED//
BENJAMIN M. COOK, Lt Col, USAF
Deputy Chief, Spacecraft Technology Division
Space Vehicles Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
09-05-2014	Final Report	28 Nov 2012 – 24 Feb 2014

4. TITLE AND SUBTITLE
The Challenges of Sensing and Repairing Software Defects in Autonomous Systems

5a. CONTRACT NUMBER
FA9453-13-1-0235

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62601F

6. AUTHOR(S)
Stephanie Forrest and Westley Weimer

5d. PROJECT NUMBER
5018

5e. TASK NUMBER
PPM00019583

5f. WORK UNIT NUMBER
EF009906

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Regents of the University of New Mexico
MSC01 1247
1 University of New Mexico
Albuquerque, NM 87131-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Space Vehicles Directorate
3550 Aberdeen Ave., SE
Kirtland AFB, NM 87117-5776

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RVSS

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RV-PS-TR-2014-0035

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A small study investigated the potential benefits and research challenges related to the software components of future space vehicle designs. The study identified two potential research thrusts aimed at improving the resilience and reliability of software deployed on space vehicles: (1) improving software resiliency through proactive diversity and (2) reducing costs and schedule overruns through automated software repair. Both thrusts rely on recently developed technology known as GenProg. GenProg uses genetic programming (GP), an iterated stochastic search technique, to search for program repairs. The search space of possible repairs is infinitely large, and GenProg employs five strategies to render the search tractable: (1) coarse-grained, statement-level patches to reduce search space size; (2) fault localization to focus edit locations; (3) existing code to provide the seed of new repairs; (4) fitness approximation to reduce required test suite evaluations; and (5) parallelism to obtain results faster. The study focused on automated software transformations for repair and resiliency, because there is extensive prior work on the related topics of anomaly detection, intrusion detection and fault isolation, which could also be adapted to software in the space vehicles domain.

15. SUBJECT TERMS
Autonomous systems software repair, Fault tolerant software systems, Resilient Software systems, satellite software self-repair, self-healing software systems, Sensing & repairing software defects in autonomous

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Leslie Vaughn
Unclassified	Unclassified	Unclassified	Unlimited	24	19b. TELEPHONE NUMBER *(include area code)*

(This page intentionally left blank)

Table of Contents

Section **Page**

ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA9453-13-1-0235. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

DISCLAIMER

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

1. Summary

A small study investigated the potential benefits and research challenges related to the software components of future space vehicle designs. The study identified two potential research thrusts aimed at improving the resilience and reliability of software deployed on space vehicles: (1) Improving software resiliency through proactive diversity and (2) reducing costs and schedule overruns through automated software repair. Both thrusts rely on recently developed technology known as GenProg. GenProg uses genetic programming (GP), an iterated stochastic search technique, to search for program repairs. The search space of possible repairs is infinitely large, and GenProg employs five strategies to render the search tractable: (1) coarse-grained, statement-level patches to reduce search space size; (2) fault localization to focus edit locations; (3) existing code to provide the seed of new repairs; (4) fitness approximation to reduce required test suite evaluations; and (5) parallelism to obtain results faster. The study focused on automated software transformations, for repair and resiliency, because there is extensive prior work on the related topics of anomaly detection, intrusion detection and fault isolation, which could also be adapted to software in the space vehicles domain.

2. Introduction

This document constitutes the final report of a one-year study to outline important software challenges facing space vehicles, in particular, the challenges of detecting and repairing software bugs on deployed vehicles.

Over the course of this grant we investigated ways that autonomous space vehicles and space vehicle software could be designed to sense and respond to software problems. In addition to scholarly activities such as publishing and presenting research results, the grant also facilitated multiple face-to-face meetings with personnel from the Space Vehicles Directorate at Kirtland Air Force Base.

3. Methods, Assumptions, and Procedures

Space vehicles and satellites are a critical component of our national and commercial infrastructures, and their autonomous operation and physical separation present special challenges for maintenance and reliability. Modern satellites are complex systems comprised of hardware and software, including sensors, actuators, special-purpose operating systems, programs and algorithms. These systems are managed using well-established techniques, such as on-board sensing, strings of redundant components, and fail-over to “safe mode” for handling certain classes of faults. Such techniques help mitigate physical faults caused by environmental conditions, such as radiation, and are not directly applicable to software faults.

Beyond physical faults, software defects are becoming a significant concern for space vehicles. For example, a recent billion-dollar deal from the United Arab Emirates to purchase two intelligence satellites from France was harmed by the discovery of two “security compromising

components" in purchased software that would provide a back door to the data transmitted to the ground station [2, 27]. Software defects, whether malicious or unintentional, are likely to become a serious problem in the future, as the complexity and functionality of deployed software on space vehicles increases. Given the high cost of a typical Geosynchronous Earth Orbit (GEO) satellite and launch vehicle, potential failures arising from software problems are worth mitigating. Historical data from 1981–2001 suggest that 9% of satellites fail during their operational lives and 4–5% of launch vehicles fail, totaling about one in seven satellites failing prematurely [35]. Looking forward, software failures are likely to increase in prominence: A June 28, 2012 report by the Department of Homeland Security noted that the number of "incidents impacting organizations that own and operate control systems associated with critical infrastructure" almost tripled from 2009 to 2010, and then increased by over a factor of four between 2010 and 2011, the last year for which complete data are available [15].

Many popular security and software engineering solutions are targeted for desktop and server environments, but modern space vehicles present a different challenge. For example, although cloud-based storage and computing has become the norm in personal computing, "stand-alone" deployment is typical for space vehicles, with a trusted base station that is not accessible through public networks. As a second example, systems are created by trusted assembly methods out of custom-made or off-the-shelf components in contrast with open-source or app-store models that allow end users to easily combine software from different sources on a single system. Space vehicles are designed to continue operating autonomously in the event that contact is lost with the ground. This set of design constraints simplifies some problems (e.g., not being on an open network reduces the threat of a remote hijacking attack), exacerbates others (expensive and intermittent communication with the base station complicates the task of system upgrades or emergency repairs), and leaves some problems unchanged (e.g., the threat of malicious "logic time bombs" or inadvertent software bugs).

Unintended bugs affecting the deployed system can leave it unresponsive. Software failures in specialized devices have both civilian and military implications, ranging from lawsuits [5] to insurgents hacking United States Air Force USAF Predator unmanned aerial vehicle feeds [9]. Unintended defects are not only possible, but common: To take one popular example, the Microsoft embedded Zune media player included a bug that turned devices into unresponsive bricks [7] affecting millions of customers [6]. The bug was a relatively simple infinite loop in a date calculation algorithm that failed to account for certain leap years. As a result, the devices appeared fine during testing but failed after deployment on January 1, 2009.

Software defects similar to the Zune bug are ubiquitous. The number of outstanding software defects typically exceeds the resources available to address them [3]. Mature software projects are forced to ship with both known and unknown bugs [23] because they lack the development resources to deal with every defect. For example, one Mozilla developer claimed that, "everyday, almost 300 bugs appear far too much for only the Mozilla programmers to handle" [4]. Once identified, bugs can be challenging to repair, leading to prolonged down time. On the Mozilla project between 2002 and 2006, half of all fixed bugs took developers over 29 days

each to fix [14]. This trend is particularly troubling in critical code—in 2006, it took 28 days on average for operating system maintainers to develop fixes for security defects [36]. A recent Cambridge University study [8] estimates that software bugs cost the global economy $312 billion per year and that one-half of software development time is spent on debugging. As the costs of faulty software have continued to rise, researchers have begun developing automated methods for detecting and repairing software bugs. We believe that this work could be adapted to the special software environment of space vehicles.

4. Results and Discussion

Over the course of this award, we investigated potential software threats to space vehicles and identified future technologies to mitigate those threats. Although some satellite software modules can be formally verified and assembled in a trustworthy way using off-the-shelf components, and stand-alone deployments may have base stations that are inaccessible to public networks, there remain several different ways that software can cause downtime or mission failure for space vehicles. Further, this can even occur with the stringent security and deployment policies already in place.

Security defenses adopted by other communities may not be present or used to their fullest advantage in space vehicle systems. For example, digital signatures [30] can be used to verify the provenance and untampered nature of code. Such signatures are common in analogous embedded systems. For example, the Sony Playstation uses digital signatures to guard third-party games run on its hardware [18]. A second example is "separation of concerns," which involves using modularity and encapsulation to limit the power of software modules and thus limit the damage that can be done if that module fails. Such a separation has been identified as a way to limit future attacks [29].

Although these techniques are potentially applicable, space vehicles has several special properties that complicate their adoption, such as limited processor speed, reduced memory, smaller storage, and power constraints. Our investigation suggested that three approaches, in particular, merit further investigation.

Sensing at the Software Level. Anomaly/intrusion detection [16] involves using software-and hardware-level metrics and sensors to establish a baseline associated with normal performance and then note when the current operating profile deviates from that acceptable envelope [12]. Many space vehicles already include sensors for anomaly detection at the hardware level, and we believe that such systems could be augmented to include sensing and monitoring software. Tradeoffs exist between costs, such as operating system support requirements, and coverage, such as the number of anomalies sensed and the number of false positives reported. We hypothesize that existing expertise in sensing temperature, radiation, battery power and similar metrics can be leveraged to help protect software.

Autonomous Software Repair. Once a software bug has been detected, either through anomaly detection or by manual reporting, it must be fixed. We hypothesize that existing approaches to software repair [21, 40] can be leveraged to allow a group of heterogeneous space vehicles and/or ground stations to attempt to fix a defect autonomously. We have evidence to suggest that such an approach is feasible for commercial off the shelf (COTS) software [21] as well as embedded systems [31, 33]. Candidate repairs produced by such techniques can be inspected by ground station developers before being deployed. In addition, in critical situations, such as a fault in the communication system software, an autonomously produced repair could serve as a last line of defense to re-establish communication.

Proactive Software Diversity. Many space vehicles already include redundant system backups, but a backup that uses exactly the same software will be vulnerable to exactly the same bugs [19, 25]. Our investigation suggests that diverse variants of critical software systems can be created automatically that are functionally equivalent but feature different implementations (e.g., variable layouts, algorithmic changes, etc.). Such diverse variants present a shifting attack surface to bugs or malware [10]. In addition, we have evidence that multiple diverse variants created in advance can serve as a shield against unknown future bugs [34].

Illustrative Example. To see how these insights might play out and interact in this domain, consider the following potential use case. Consider a deployed space vehicle with three strings of redundant systems, each of which is slightly different as a result of proactive software diversity. When a software bug or piece of malware attempts to influence the first string, that deviation from the norm is likely to be sensed at the software level by anomaly/intrusion detection techniques. Control can then be transferred to the second string of systems, which are not vulnerable to the same bug because of their different attack surface. The second string and the ground system can then work together to patch using autonomous software repair, and that fixed replacement software can then be uploaded or deployed over the old first string software system.

Challenges Identified. We held several meetings between the investigators and the space vehicles community, and we identified the following challenges for this domain.

1. Relatively low processing power on the space vehicle and high processing power on ground. The processors that fly in space are many generations behind state-of-the-art technology available in the consumer market. This is not surprising given the extraordinary testing and hardening that must be performed on hardware before deployment.
2. Separation between operating system and payload software. This provides a possible opportunity to apply proactive diversity and automated repair methods to the operating system without interfering with payloads, or vice-versa.
3. Fail-over redundancy structure is already commonly used. This is a stark contrast to standard desktop computing.
4. Power consumption and heat dissipation matter. Recent results on post-compiler

optimizations to reduce energy use of software [32] may be applicable to address this challenge.

5. Users desire high reliability, but systems are often assembled near the deadline using off-the-shelf-components. An ability to operate through errors using automatic software repair methods would help address this challenge.
6. Users desire high uptime, and a "degraded mode" response to faults may be preferable to a "fail-stop" response.
7. Computational resources for sandboxing and evaluating variants (i.e., candidate repairs) are limited.

4.1 Technical Approach

Over the course of this grant and previous awards we have developed a technique for automatically repairing software defects in off-the-shelf, legacy programs. We call this approach GenProg, and it has scaled to repair defects in software totaling five million lines of code guarded by ten thousand test cases [21, 22, 40]. The basic operation of GenProg on desktop software is described in Section 4.5, which serves as essential background for understanding changes that might be made to apply such a system to space vehicles.

Key capabilities that are relevant to the space vehicles domain are: (1) the ability to automatically repair classes of software bugs that are not pre-specified; (2) the ability to generate multiple semantically distinct program variants, each of which meets an existing program specification (either formally defined or implicitly defined through test cases); and (3) the ability to apply heuristic transformations to compiled code (at the assembly level or binary level) to reduce energy consumption, or to improve other nonfunctional software properties.

4.2 Promising Future Research Directions

We identified two promising threads for future research: Improving resiliency through proactive diversity, and reducing cost through automated repair.

4.2.1 Basic Research to Improve Resiliency

To improve the resiliency of software deployed on space vehicles, we propose developing proactive software diversity methods that are practical for software systems facing the challenges outlined above. We propose to first measure the mutational robustness of the relevant software and then develop methods for automatically generating multiple semantically distinct software versions. We envision that several of these diverse versions would be deployed on a single space vehicle.

Our approach recognizes that only attackers (e.g., buffer overruns) and software bugs (e.g., infinite loops) depend on under-the-hood implementation behavior. For example, while buffer

overruns depend strongly on the order in which the compiler lays out variables on the stack, legitimate use cases do not, and thus a variant that re-orders the stack may defeat attackers without reducing functionality.

Motivation: Satellite failure rates are too high. 9% of satellites fail during operational lives, and 4–5% of launch vehicles fail, for a total of about 1-in-7 that fail prematurely [35]. Fail-over redundancy for software only protects against transient errors (e.g., radiation bit-flips), but not against most program bugs or logic bombs [25]. For example, if third-party COTS software has a bug and always fails after January 1, 2009 (as in the infamous Microsoft Zune player), failing over to an identical copy results in a system that immediately encounters the same bug.

Proposed Research Activities:

1. Develop techniques to automatically generate diverse variants of payload (or control and payload) software for space vehicles.

2. Develop algorithms to generate software variants that implement the same specification but have multiple invisible implementation differences "under the hood" (e.g., scanning left-to- right instead of right-to-left).

3. Construct a system in which multiple generated software variants present a shifting defensive surface and are vulnerable to different failures. If even one is resilient, the system can enter safe mode and contact the base station:

    (a) For example, consider a situation in which a software defect akin to the "January 1, 2009" Zune bug causes the first-string space vehicle software to fail. If the second-string software is not identical, but is instead a variant that uses different implementation decisions, failing over to the second-string could resolve the issue if the second-string software were immune to the bug (e.g., because it handled the date calculation loop differently).
    (b) In addition, such an approach would retain all expected resilience to transient "bit-flip"- style errors.

Why Now, Why Here?
Space vehicles are an ideal setting to develop proactive diversity techniques, because it is well-understood that investing in redundancy can avoid some failures, and this trade-off is accepted in the community. By contrast, in standard software engineering, companies are rarely willing to buy a second or third set of completely redundant hardware. Techniques for generating semantically equivalent diverse variants have only recently become available [34].

Basic Research Questions:

1. How can we automatically and efficiently generate a large number of diverse variants of a software program? There are three distinct issues to be addressed: (1) Determining what program representation is most appropriate (abstract syntax trees, assembly code, object code); (2) determining which mutational operators should be used to generate the diversity (delete, swap, replace, copy, etc.); and (3) determining how to evaluate if the variations meet the desired specification (e.g., test cases, formal specifications, user interaction, etc.).
2. How can we prove (or gain evidence) that these variants are not vulnerable to the same faults (independent failure modes)? For example, we envision using fault injection techniques, "time travel" studies of historical data, static analyses techniques, or predictive fault models.
3. Given that we only have space to deploy k fail-over backups, how should the k variants be chosen to maximize deployed diversity (i.e., maximize the chance that at least one will defeat a new fault)? We propose using diversity distance metrics (including advanced information flow techniques) or clustering algorithms.
4. A more ambitious research topic would investigate how to select variants for diversity and to minimize power and/or memory use (software-only schemes can reduce software power use 13–40% [38]).

4.2.2 Basic Research to Reduce Costs and Schedule Overruns

Software maintenance is an ongoing expense, which could be reduced if some maintenance tasks were automated. We propose to focus on repairing bugs in software, first in the pre-deployment phase, and as a long-term goal, to repair software that has already been deployed.

Motivation: Crafting and validating patches for software bugs can take ground teams weeks to months for space vehicles, and the space vehicle payload may be disabled in safe mode while awaiting the repair.

Proposed Research Activities:

1. Develop and refine techniques to automatically generate software patches using genetic programming.
2. Design automated repair algorithms such that, by construction, synthesized patches address the defect while retaining all tested functionality.
3. Generate a diverse set of candidate patches and present them to ground developers:
   (a) Previous human studies have demonstrated that developers presented with machine- generated patches take less time to address defects [39] and that such patches can be as readable and maintainable as human-written patches [13].
   (b) Multiple independent, differently shaped patches will help developers catch all

corner cases (e.g., if one patch fixes the definition of foo and another fixes all uses of foo, developers can adapt, merge or augment the suggestions).

(c) Patches can be constructed to minimize verification effort (e.g., favoring patches that touch the fewest modules, have minimal change impact, etc.) or otherwise integrate well with formal methods [28, 41, 42].

4. Develop techniques to use only some of the tests when "brainstorming" candidate patches and use all of the tests only to verify those that make the final cut before showing them to developers.

Why Now, Why Here?

Modern hardware (e.g., clusters, cloud computing) is such that computers can now generate and evaluate patches faster than humans. Many reported software bugs for space vehicles (e.g., crashing, excessive memory usage, infinite loops) are amenable to preliminary single- patch genetic programming techniques [22]. In a systematic study, our proposed GenProg approach generated a single working patch for 50% of desktop software bugs for one-third the cost of human developers [21].

Basic Research Questions:

1. How can we develop benchmark programs and bugs relevant to the space vehicles community (i.e., where software is meaningfully different from previously studied web browsers and databases) that will allow us to measure success?
2. How can we generate multiple informative, instructive patches to space vehicle software defects?
3. How can we generate patches with reduced verification burdens?
4. Can we develop techniques to rapidly construct circumscribed repairs that isolate and leave available some payload behavior or modules while walling off and shutting down others? The goal is to develop an expanded safe mode in which some prescribed payload functions remain usable while awaiting the final patch.

5 Automated Program Repair

We describe the basic operation of GenProg on desktop software, which serves as essential background for understanding the research that would be required to apply such a system to space vehicles.

5.1 Genetic Programming

GenProg uses genetic programming (GP) [20], an iterated stochastic search technique, to search for program repairs. The search space of possible repairs is infinitely large, and GenProg employs five strategies to render the search tractable: (1) coarse-grained, statement-level patches to reduce search space size; (2) fault localization to focus edit locations; (3) existing code to

provide the seed of new repairs; (4) fitness approximation to reduce required test suite evaluations; and (5) parallelism to obtain results faster.

GenProg's main algorithm takes the form of an iterative loop to construct and evaluate fit patches. Fitness is measured by counting the number of test cases passed by a candidate repair. The goal is to produce a candidate patch that causes the original program to pass all test cases, including those that encode the defect. We represent each candidate patch [1] as a sequence of abstract syntax tree (AST) edit operations parameterized by node numbers (e.g., Replace (81, 44)); see Section 5.2).

Given a program and a test suite, we localize the fault (Section 5.4) and compute context-sensitive information to guide the search for repairs (Section 5.5) based on program structure and test case coverage. We evaluate variant fitness (Section 5.3) by applying candidate patches to the original program to produce a modified program that is evaluated on test cases. New candidate patches are constructed from existing patches via mutation and crossover operators defined in Section 5.6 and Section 5.7. Both generate new patches to be tested.

The search begins by constructing and evaluating a population of random patches. We initialize a population by independently mutating copies of the empty patch. In each generation (iteration) we employ tournament selection [26], which selects from the incoming population, with replacement, high-fitness parent individuals. By analogy with genetic "crossover" events, parents are taken pairwise at random to exchange pieces of their representation; two parents produce two offspring (Section 5.7). Each parent and each offspring is mutated once (Section 5.6) and the result forms the incoming population for the next iteration. The GP loop terminates if a variant passes all test cases, or when resources are exhausted (i.e., too much time or too many generations elapse). We refer to one execution of this algorithm as a trial. Multiple trials may be run in parallel, each initialized with a distinct random seed.

The rest of this section describes additional algorithmic details, including: (1) a patch-based representation, (2) large-scale use of a sampling fitness function at the individual variant level, (3) fix localization to augment fault localization, (4) and novel mutation and crossover operators to dovetail with the patch representation.

5.2 Patch Representation

An important GenProg enhancement involves the choice of representation. Each variant is a patch, represented as sequence of edit operations (compare to [1]). It is possible to represent an individual by its entire AST combined with a weighted execution path [40], but such an approach does not scale to memory-constrained environments. For example, for one- third of defects we have considered experimentally, a population of 40–80 ASTs did not fit in 1.7 GB of main memory. However, half of all human-produced software patches are 25 lines or less [21]. Thus, two unrelated variants might differ by only 2 × 25 lines, with all other AST nodes in common.

Representing individuals as patches avoids storing redundant copies of untouched lines. This formulation influences the mutation and crossover operators, discussed below.

5.3 Fitness Evaluation

To evaluate the fitness of a large space of candidate patches efficiently, we exploit the fact that GP performs well with noisy fitness functions [11]. For intermediate calculation, we apply a candidate patch to the original program and evaluate the result on a random sample of the tests, choosing a different test suite sample each time. For efficiency, only variants that pass every test in the sample are fully tested on the entire test suite. The final fitness of a variant is the sum of the number of tests that are passed.

5.4 Fault Localization

GenProg focuses repair efforts on statements likely to be implicated in the defect. Such fault localization approaches are well-established in software engineering [17]. For a given program, defect, set of tests T, test evaluation function Pass: T → B, and set of statements visited when evaluating a test Visited: T → P(Stmt), we define the fault localization function faultloc : Stmt → R to be:

$$\text{faultloc}(s) = \begin{cases} 0 & \forall t \in T.\ s \notin \text{Visited}(t) \\ 1.0 & \forall t \in T.\ s \in \text{Visited}(t) \implies \neg \text{Pass}(t) \\ 0.1 & \text{otherwise} \end{cases}$$

That is, a statement never visited by any test case has zero weight, a statement visited only on a bug-inducing test case has high (1.0) weight, and statements covered by both bug-inducing and normal tests have moderate (0.1) weights (this strategy follows previous work [40, Sec. 3.2]). Other fault localization schemes could be employed directly by GenProg [24].

5.5 Fix Localization

We introduce the term fix localization (or fix space) to refer to the source of insertion/replacement code, and explore ways to improve fix localization beyond blind random choice. As a start, we restrict inserted code to that which includes variables that are in-scope at the destination (so the result compiles) and that are visited by at least one test case (because we hypothesize that certain common behavior may be correct). For a given program and defect we define the function fixloc : Stmt → P(Stmt) as follows:

$$\text{fixloc}(d) = \{\, s \mid \exists t \in T.\ s \in \text{Visited}(t) \wedge \text{VarsUsed}(s) \subseteq \text{InScope}(d) \,\}$$

The fix localization function just defined helps to ensure that candidate patches are well-formed:

in our experiments, more than 90% of candidates compile correctly.

5.6 Mutation Operator

We consider three mutation operators: delete, insert and replace. In a single mutation, a destination statement d is chosen from the fault localization space (randomly, by weight). With equiprobability GenProg either deletes d (i.e., replaces it with the empty block), inserts another source statement s before d (chosen randomly from fixloc(d)), or replaces d with another statement s (chosen randomly from fixloc(d)). Inserted code is taken exclusively from elsewhere in the same program. This decision reduces the search space size by leveraging the intuition that programs contain the seeds of their own repairs.

5.7 Crossover Operator

The crossover operator combines partial solutions, helping the search avoid local optima. Our new patch subset crossover operator is a variation of the well-known uniform crossover operator [37] tailored for the program repair domain. It takes as input two parents, p and q, represented as ordered lists of edits (Section 5.1). The first (resp. second) offspring is created by appending p to q (resp. q to p) and then removing each element with independent probability of one-half. This operator has the advantage of allowing parents that both include edits to similar ranges of the program (e.g., parent p inserts B after A and parent q inserts C after A) to pass any of those edits along to their offspring. Previous uses of a one-point crossover operator on the fault localization space did not allow for such recombination (e.g., each offspring could only receive one edit to statement A).

5.8 Binary and Assembly Repairs

The initial versions of GenProg focused on abstract syntax tree representations of C programs. More recently, we have developed a prototype implementation for the Low Level Virtual Machine (LLVM) compiler suite where the program is represented using LLVM's intermediate representation, and the operators are defined over the intermediate representation.

We have also developed technology for compiled (ARM and x86 assembly) and linked Execute and Linkable Format (ELF) binary programs [33, 31]. The new representations allow repairs when source code cannot be parsed into ASTs (e.g., due to unavailable source files, complex-build procedures, or non-C source languages). They also reduce memory and disk requirements sufficiently to enable repairs on resource constrained devices.

Relevant to the space vehicles domain, our techniques have been shown to reduce memory requirements by up to 85%, disk space requirements by up to 95%, and repair generation time up to 62%, which enables application to resource-constrained environments.

These techniques constitute the first general automated method of program repair applicable to binary executables and are an important first step towards on-board repair of software defects on space vehicles where memory and computation resources are limited.

6 Conclusion

This report describes the work of a short preliminary study to explore the challenges of sensing and repairing software defects in autonomous systems. We focused on the repair challenge because there is extensive prior work on anomaly detection, intrusion detection, and fault isolation which could be adapted to this domain.

We identified two promising areas for future research projects and outlined our thoughts about how best to pursue them: (1) Improving software resiliency through proactive diversity and (2) reducing costs and schedule overruns through automated software repair.

References

[1] T. Ackling, B. Alexander, and I. Grunert. "Evolving patches for software repair." *Genetic and Evolutionary Computation*, pp. 1427–1434, 2011.

[2] I. Allen. "Discovery of spy parts leaves french-UAE satellite deal in doubt," URL: IntelNews at http:// intelnews. org/ 2014/ 01/ 07/ 01-1402/, Jan. 2014.

[3] J. Anvik, L. Hiew, and G. C. Murphy. "Coping with an open bug repository," In OOPSLA Workshop on Eclipse Technology eXchange, pp. 35–39, 2005.

[4] J. Anvik, L. Hiew, and G. C. Murphy. "Who should fix this bug?" International Conference on Software Engineering, pp. 361–370, 2006.

[5] M. Barr. "Faulty code will lead to an era of firmware-related litigation," *In Electronic Design*, Jan. 2010.

[6] D. Bass. "Microsoft updates Zune devices, pledges to gain sales." URL: http://www.bloomberg.com/apps/news?pid=newsarchive&sid=aKNQROlvcaOM, Bloomberg, Oct. 2007.

[7] BBC News. "Microsoft Zune affected by 'bug'" URL: http: // news. bbc. co. uk/ 2/ hi/ technology/ 7806683. stm, Dec. 2008.

[8] G. Carver, L. Jeng, Paul, Cheak, T. Britton, and T. Katzenellenbogen. "Experts battle £192bn loss to computer bugs." URL: http://www.cambridge-news.co.uk/Education/Universities/Experts-battle-192bn-loss-to-computer-bugs-18122012.htm, 2012.

[9] Mike Mount, Quijano Elaine. "Iraqi insurgents hacked Predator drone feeds, U.S. official indicates" CNN.com, URL: http: // www. cnn. com/ 2009/ US/ 12/ 17/ drone. video. hacked/ index. html , Dec. 17, 2009.

[10] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. "N-variant systems: a secretless framework for security through diversity." *USENIX Security Symposium*, Vancouver, B.C., Canada, 2006.

[11] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. "Designing better fitness functions for automated program repair." *Genetic and Evolutionary Computation Conference*, Portland, Oregon, pp. 965–972, 2010.

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. "A sense of self for Unix processes." *IEEE Symposium on Security and Privacy*, Oakland, CA, pp. 120–128, 1996.

[13] Z. P. Fry, B. Landau, and W. Weimer."A human study of patch maintainability." *International Symposium on Software Testing and Analysis*, Minneapolis, MN, pp. 177–187, 2012.

[14] P. Hooimeijer and W. Weimer. "Modeling bug report quality." *Automated Software Engineering*, pp. 34–43, 2007.

[15] ICS-CERT incident response summary report. "Industrial Control Systems Cyber Emergency Response Team." US Department of Homeland Security, 13-50012, URL: http://www.uscg.mil/hq/cg5/cg544/docs/Year_in_Review_FY2012_Final.pdf , 2012.

[16] K. L. Ingham and A. Somayaji. "A methodology for designing accurate anomaly detection systems." *IFIP/ACM Latin American Networking Conference*, San Jose, Costa Rica, 2007.

[17] J. A. Jones and M. J. Harrold. "Empirical evaluation of the Tarantula automatic fault-localization technique." *In Automated Software Engineering*, pp. 273–282, 2005.
[18] J. Kirk. "Sony asks for restraining order over PS3 hack." URL: http://www.computerworld.com/s/article/9204723/Sony_asks_for_restraining_order_over_PS3_hack, Government IT, last modified Jan. 2011. Accessed March 31, 2014.
[19] J. C. Knight and P. Ammann. "Issues influencing the use of n-version programming." *IFIP Congress*, San Francisco, CA, 1989.
[20] J. R. Koza. "Genetic Programming: On the Programming of Computers by Means of Natural Selection." *MIT Press*, 1992.
[21] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each." International Conference on Software Engineering, Zürich, Switzerland, pp. 3–13, 2012.
[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. Transactions on Software Engineering, 38(1):54–72, 2012.
[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. "Bug isolation via remote program sampling." Programming Language Design and Implementation, pp. 141–154, 2003.
[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. Programming Language Design and Implementation, pp. 15–26, 2005.
[25] D. E. Lowell, S. Chandra, and P. M. Chen. "Exploring failure transparency and the limits of generic recovery." *USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.
[26] B. L. Miller and D. E. Goldberg. "Genetic algorithms, selection schemes, and the varying effects of noise." Evolutionary Computing, **4**(2):113–131, 1996.
[27] A. Mustafa and P. Tran. "French-UAE intel satellite deal in doubt." DefenseNews URL: http: // www. defensenews. com/ article/ 20140105/ DEFREG04/ 301050006, Jan. 2014.
[28] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. "SemFix: Program repair via semantic analysis." *International Conference on Sofware Engineering*, pp. 772–781, 2013.
[29] J. Nielsen. "Building secura applications." Technical report, Trapeze Group Europe A/S, URL: http://www.broadband-testing.co.uk/download/TrapezeWLAN41v1.pdf , June 2006.
[30] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-keycryptosystems." Commun. ACM, **21**(2): pp.120–126, Feb. 1978.
[31] E. Schulte, J. DiLorenzo, S. Forrest, and W. Weimer. "Automated repair of binary and assembly programs for cooperating embedded devices." *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, 2013.
[32] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. "Post-compiler software optimization for re- ducing energy." *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, 2014.
[33] E. Schulte, S. Forrest, and W. Weimer. "Automatic program repair through the evolution of assembly code." Automated Software Engineering, pp. 33–36, 2010.
[34] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. "Software mutational robustness." Genetic Programming and Evolvable Machines, pp. 1–32, 2013.

[35] B. Sullivan and D. Akin. "A survey of serviceable spacecraft failuers." American Institute of Aeronautics and Astronautics, 2001(4540):1–8, 2001.
[36] "Symantec Internet security threat report." URL: http: // eval. symantec. com/ mktginfo/ enterprise/ white_ papers/ ent-whitepaper_ symantec_ internet_ security_ threat_ report_ x_ 09_ 2006. en-us. pdf, Vol. X, Sept. 2006.
[37] G. Syswerda. "Uniform crossover in genetic algorithms." J. D. Schaffer, editor, *International Conference on Genetic Algorithms*, San Francisco, CA, pp. 2–9, 1989.
[38] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: a first step towards software power minimization." IEEE Trans. VLSI Syst., **2**(4):437–445, 1994.
[39] W. Weimer. "Patches as better bug reports." Generative Programming and Component Engineering, pp. 181–190, 2006.
[40] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. "Automatically finding patches using genetic programming." International Conference on Software Engineering, Vancover, B.C, Canada, pp. 364–367, 2009.
[41] X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer. "Formal verification by reverse synthesis." Computer Safety, Reliability, and Security, pp. 305–319, 2008.
[42] X. Yin, J. C. Knight, and W. Weimer. "Exploiting refactoring in formal verification." International Conference on Dependable Systems and Networks, Estoril, Lisbon Protugal, pp. 53–62, 2009.

LIST OF ACRONYMS

ARM	Abstract Rewriting Machine (also a compiler infrastructure)
AST	Abstract Syntax tree
COTS	Commercial of the Shelf
ELF	Executable and Linkable Format
GenProg/GP	Genetic Programming
GEO	Geosynchronous Earth Orbit
LLVM	Low Level Virtual Machine (this is a compiler infrastructure)
USAF	United States Air Force
X86	Family of backward compatible instruction set architectures based on the Intel 8086(Intel Corp part number) central processing unit

DISTRIBUTION LIST

DTIC/OCP
8725 John J. Kingman Rd, Suite 0944
Ft Belvoir, VA 22060-6218 1 cy

AFRL/RVIL
Kirtland AFB, NM 87117-5776 2 cys

Official Record Copy
AFRL/RVSS/Leslie Vaughn 1 cy

(This page intentionally left blank)